AD-786 723

LCFsmall: AN IMPLEMENTATION OF LCF

Luigia Aiello, et al

Stanford University

Prepared for:

Office of the Secretary of Defense
Advanced Research Projects Agency

August 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-74-446	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-786 723
4. TITLE (and Subtitle) LCFsmall: AN IMPLEMENTATION OF LCF		5. TYPE OF REPORT & PERIOD COVERED technical, Aug. 1974
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-74-446 and AIM 241
7. AUTHOR(s) L. Aiello and R. W. Weyhrauch		8. CONTRACT OR GRANT NUMBER(s) DAHC 15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Department Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA 2495
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT, Attn: Stephen D. Crocker, 1400 Wilson Blvd., Arlington, Va. 22209		12. REPORT DATE Aug. 1974
		13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative, Philip Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitations on dissemination.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a report on a computer program implementing a simplified version of LCF. It is written (with minor exceptions) entirely in pure LISP and has none of the user oriented features of the implementation described by Milner. We attempt to represent directly in code the metamathematical notions necessary to describe LCF. We hope that the code is simple enough and the metamathematics is clear enough so that properties of this particular program (e.g. its correctness) can eventually be proved. The program is reproduced in full.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM No.241

AUGUST 1974

COMPUTER SCIENCE DEPARTEMENT REPORT
STAN CS 74-446

LCFsmall: an implementation of LCF

by
Luigia Aiello
and
Richard W. Weyhrauch

Abstract:

This is a report on a computer program implementing a simplified version of LCF. It is written (with minor exceptions) entirely in pure LISP and has none of the user oriented features of the implementation described by Milner. We attempt to represent directly *in code* the metamathematical notions necessary to describe LCF. We hope that the code is simple enough and the metamathematics is clear enough so that properties of this particular program (e.g. its correctness) can eventually be proved. The program is reproduced in full.

Authors' addresses

L. Aiello, Istituto di Elaborazione dell'Informazione, via S. Maria 46, 56100 Pisa, Italy;

R. Weyhrauch, A.I. Lab. Computer Science Dept., Stanford University, Stanford, California 94305.

This research is supported (in part) by the Advanced Research Project Agency of the Office of the Secretary of Defense (DAHC 15-73-C-0435).

LCFsmall

TABLE OF CONTENTS

SECTION 1 Introduction

LCFsmall is a case study. It was designed to shed light on several aspects of current research in the mathematical theory of computation and representation theory. As a side benefit it is a program which can be used to do experiments using the typed λ-calculus to interpret programming languages. This approach was first discussed by D. Scott in 1969. For us it was also an exercise in writing such a system without the aid of the MLISP2 extendible parser (Smith and Enea 1973).

LCFsmall is an implementation of a proof-checker for the unadorned logical calculus. LCF itself augments this basic logic with additional rules and user aids in an attempt to make the actual checking of proofs more feasible. These include the simplification rule, a facility for using theorems, and the subgoal structure. LCFsmall has an entirely different motivation. First, a natural question about LCF has always been "*but who checks the checker?*", i.e. have you *proved* that LCF is *correct*? This task is simply too big to be considered given our present capabilities for proving the correctness of programs. LCF uses backtracking and is about 35 pages of MLISP2 code. With no extra free storage, it is a 48K (PDP10 36 bit word) program. We think that is the long run the reliability (or correctness if you wish) of such large programs needs to be considered.

Several things happened to make us look at this task at different levels. First we had learned a lot about constructing proof checkers while experimenting with LCF and a new *cleaned up* version was envisioned. Secondly, M. Newey 1974 has presented an LCF axiomatization of LISP, and done several extremely large proofs. This led us to consider the idea of writing a new version of LCF entirely in LISP, which had some hope of being proved correct. Moreover, using pure LISP increases its portability. In actual fact it is written and printed here in MLISP2. The translation into pure LISP, however, is straight forward and we felt this was easier to read. A copy of the LISP code can be gotten by writing to Richard Weyhrauch.

In order that a proof of correctness be at all feasible we decided only to include those rules originally suggested by D. Scott in 1969. These are explained in detail in Milner 1972 and Weyhrauch and Milner 1972. For the purpose of this note we expect familiarity with one of these papers.

Another motivation was our interest in seeing just how straightforward it was to translate the "metamathematical description" of LCF directly into code. That is we tried to write the program in terms of the notions involved.

A typical metamathematical description of a logical calculus involves some general inductive definitions of sentences in the language, together with a description of the rules and an inductive definition of derivations. These definitions suggest code directly. A reasonable question is: is this "code" usable and does it do the job, i.e. is it correct? The problem of changing inductive definitions (i.e. most frequently context free grammars of one sort or another) into parsers has been discussed a lot. We do not go into it here. One result of this work, however, was the recognition for a kind of control structure which we would have found very helpful. It is related to the notion of updaters for data structures (see Hoare 1973).

Consider the following description of substitution of a term t for a variable v, in an expression e.

```
subst(t,v,e) ≡ IF isfreefor(t,v,e) THEN replace(t,v,e) ELSE e

isfreefor(t,v,e) ≡ IF atomic(e) THEN true
                   ELSE IF isquantwff(e)
                        THEN IF boundvarof(e)=v THEN true
                             ELSE IF boundvarof(e)∊freevarof(t)∧occursfreein(v,e) THEN false
                             ELSE ∀x∊PART(e).isfreefor(t,v,x)
                   ELSE ∀x∊PART(e).isfreefor(t,v,x)

occursfreein(v,e) ≡ IF v=e THEN true
                    ELSE IF atomic(e) THEN false
                    ELSE IF isquantwff(e)∧boundvarof(e)=v THEN false
                    ELSE ∃x∊PARTS(e).occursfreein(v,x)

replace(t,v,e) ≡ IF v=e THEN t
                 ELSE IF atomic(e) THEN e
                 ELSE REBUILD e USING replace(t,v,x) FOR x∊PARTS(e)
```

This code is almost a direct translation of the first order description of the notions involved. However, there appear constructs which are not generally available in existing programming languages and are not implementable simply or efficiently by a macro facility.

Consider for example the following four constructs:

```
∀x∊A.B[x]
∃x∊A.B[x]
PARTS(e)
REBUILD e USING F(x) FOR x∊PARTS(e)
```

Each of them represents a kind of mapping function on different data structures.

```
∀x∊A.B[x]
```

is interpreted as: if A is a "set" then for each element of A, bind it to x and evaluate B. When you are finished return the value of the conjunction of the results. In MLISP2 this function can be realized by

```
FOR NEW X IN A DO :AND B[X]
```

but we do not use this construct in the code below as its translation into LISP is not immediate.

```
∃x∊A.B[x]
```

is the same as above replacing disjunction for conjunction.

The other two constructs are more difficult as they require a new look at the definition of data structures. For PARTS(e), the program must be able to decide what kind of thing e is, and how to canonically take it apart. In our example REBUILD returns the homomorphic image of e with respect to replace and the basic constructors of e. This type of updating data structures is considered in Hoare 1973.

The above examples show that the direct translation of metamathematics into code requires programming language features not yet generally available, and show that these features arise naturally in applications. These examples of course do not use assignment statements to "remember" certain facts and possibly are computed several times, making this code inefficient. We do not believe, however, that it is too bad. This kind of redundant computation can be detected by a compiler.

The code below is a compromise using only those features available in pure LISP, rather than defining these constructs in LISP and then writing code in terms of them.

In all cases the code has been written abstract syntactically and the actual data structures are not mentioned. The ones we have chosen are found in appendix 7.

SECTION 2 Description of LCFsmall

In this section we describe LCFsmall and compare it with LCF as described in Milner 1972. In LCFsmall no restriction has been imposed on the logic, all the inference rules described in Milner 1972, section 2 are included in it. On the contrary, restrictions have been imposed on the commands. LCFsmall has none of the facilities included in LCF to help the user in making proofs. It has no subgoaling mechanism, no simplifications facilities, no possibility of declaring axioms and using theorems. Steps of the proofs cannot be labeled, so the only way of referencing them is by their stepnumber. Proofs can only be carried out by a forward deduction without any abbreviation. In addition, restrictions have been imposed on the syntax of terms. In LCFsmall parentheses can never be omitted.

LCF has no CASES and INDUCT commands, because the corresponding subgoaling tactics are more useful in making proofs. We have included these commands in LCFsmall since it has no subgoaling mechanism. Moreover, LCFsmall has a ALPHACONV command absent in LCF. It is used for changing names to bound variables. This command is not included in LCF, since it automatically renames conflicting variables.

Section 2.1 Inference commands

In the description of commands, as well as in the code presented in the appendices, the following metavariables will be used:

L, L1, L2... denote stepnumbers,

N, N1, N2... denote nonnegative integers,

V, V1, V2... denote identifiers,

TRM, TRM1... denote terms.

AWF, AWF1... denote atomic well formed formulas (awff),

WF, WF1... denote well formed formulas (wff),

To facilitate the comparison with LCF, commands are listed in the same order as in Milner 1972. As a general remark, note that commas are never used as delimiters in LCFsmall, blanks are used instead.

Without worrying about the data structure (it will be described in 3.6) we note that a LCF proof is a sequence of steps. Each of them is generated by one of the following commands and it consists of a stepnumber, a wff (possibly consisting of only one awff), the list of stepnumbers it depends upon, and the reason, i.e. the command by which it has been obtained.

ASSUME AWF,

generates a new step in the proof. The AWF is added to the proof as a new step depending on itself.

INCL L1 N;

generates a new step whose awff is the N-th awff in the step L1, and whose dependencies are the same as L1.

CONJ L1 L2;

the wffs in L1 and L2 are unioned and put in a new step whose dependencies are the union of those of L1 and L2.

CUT L1 L2;

if L1 and L2 are steps in the proof and if each awff appearing in the dependencies of L2 appear in L1, then a new step is generated. Its dependencies are those of L1 and its wff is that of L2;

HALF L1;

If the first awff in L1 contains the "≡" symbol, then a new step is generated. Its awff is obtained from the first awff of L1 replacing "≡" by "⊂". The dependencies of the new step are those of L1.

SYM L1;

This command is similar to the previous one. In this case the two terms of the first awff in L1 are interchanged.

TRANS L1 L2;

If the first awff in L1 is of the form TRM1≡TRM2 and the first awff in L2 has the form TRM2≡TRM3, a new step is generated. Its awff is TRM1≡TRM3 and its dependencies are the union of those of L1 and L2. If in one (or both) of the above awffs the symbol "⊂" appears, then "⊂" will appear in the new step.

APPL L1 TRM;

APPL TRM L1;

In the first case, both sides of the first awff of L1 are applied to TRM. In the second case TRM is applied to both sides of the first awff of L1. The dependencies of the new step are those of L1.

ABSTR L1 V;

If V is an identifier not occurring free in the dependencies of L1, then a λ-abstraction is done on both terms of the first awff of L1. The dependencies of the new step are those of L1.

CASES L1 L2 L3 TRM;

Given 3 stepnumbers L1, L2 and L3 with the same wff, if one of the dependencies of L1 is TRM≡TT, one of the dependencies of L2 is TRM≡UU and one of the dependencies of L3 is TRM≡FF, then a new step is generated. Its wff is that of L1 and its dependencies are those of L1, L2 and L3 after having removed the three above dependencies regarding TRM.

INDUCT L1 L2 L3 L4 V1;

Given four stepnumbers L1, L2, L3 and L4, if the first awff of L1 is a fixpoint definition, i. e. if it has the form FIX≡[αG.FUN(G)], if the wff of L2 is obtained replacing UU for V1 in the wff of L3, if the wff of L4 is obtained replacing FUN(V1) for V1 in the wff of L3, and moreover, L3 appears in the dependencies of L4, then a new step is generated. Its wff is obtained replacing FIX for V1 in the wff of L3. The command fails if one of the above conditions is not met or if there is some variable conflict in one of the substitutions. The dependencies of the new step are the union of those of L1, L2, L3 and L4, minus L3.

CONV L1;

CONV TRM;

The conversion command has two forms: in the first one it takes a stepnumber L1 as argument. In this case, both terms of the first awff of L1 are converted and the resulting awff becomes a new step in the proof. Its dependencies are those of L1. If the argument of CONV is a term TRM a new step without dependencies is generated. Its awff is TRM≡CONVT(TRM). CONVT is a function which converts terms. Its definition is given in appendix 6.3. LCFsmall has no automatic mechanism for changing the names of conflicting bound variables. If there is some variable conflict, λ-conversions aren't performed. So the term [λy.[λx.y(x)]](x) is not converted in LCFsmall, while it is converted to [λx1.x(x1)] in LCF.

ETACONV TRM;

TRM is etaconverted. Suppose TRM has the form [λx.F(x)] with x not free in F, then a new step is generated, without dependencies, whose awff is [λx.F(x)]≡F.

ALPHACONV L1 V1 V2;

ALPHACONV TRM V1 V2;

If the first argument of ALPHACONV is a stepnumber L1, then V1 replaces V2 in its first bound occurrence in the first awff of L1. The resulting awff is put in a new step whose dependencies are those of L1. If the first argument is a term, then a new step is generated, without dependencies. Its awff is TRM≡TRM1, where TRM1 is obtained from TRM by replacing V1 for V2 in its first bound occurrence.

EQUIV L1 L2;

Given two step numbers L1 and L2 if the first awff of L1 has the form TRM1⊂TRM2 and the first awff of L2 has the form TRM2⊂TRM1, then a new step is generated. Its awff is TRM1≡TRM2 and its dependencies are the union of those of L1 and L2.

REFL1 TRM;

REFL2 TRM;

The first command generates a new step whose awff is TRM≡TRM, without any dependency. The awff generated by the second command is TRM⊑TRM.

MIN1 TRM;

MIN2 TRM

In the first case a new step is generated, without dependencies, whose awff is UU⊑TRM. In the second case the awff is UU(TRM)≡UU.

CONDT TRM;

If TRM has the form TT→TRM1,TRM2 then CONDT generates a new step whose awff is TRM≡TRM1 with no dependency.

CONDF TRM;

If TRM has the form FF→TRM1,TRM2 then CONDF generates a new step whose awff is TRM≡TRM2 with no dependency.

CONDU TRM;

If TRM has the form UU→TRM1,TRM2 then CONDU generates a new step whose awff is TRM≡UU with no dependency.

FIXP L1;

If the first awff in L1 is a fixpoint definition, i.e. if it is of the form FIX≡[αG.FUN(G)], and if FIX may be substituted for G in FUN(G) without variable conflicts, then a new step is generated. Its awff is FIX≡FUN(FIX) and its dependencies are those of L1.

SUBST L1 OCC N IN L2;

SUBST L1 OCC N IN TRM;

SUBST has two forms. In the first one, if the first awff of L1 is TRM1≡TRM2, then TRM2 is replaced for the N-th free occurrence of TRM1 in the firt awff of L2. The resulting awff is put in a new step, whose dependencies are the union of those of L1 and L2.

In the second form the command SUBST operates on a TRM. If the above hypotheses hold for L1, a new step is generated. Its dependencies are those of L1 and its awff is TRM≡SUBSTTT(TRM1,TRM2,TRM,N). The function SUBSTTT, defined in appendix 6.3, substitutes TRM2 for the N-th free occurrence of TRM1 in TRM.

Section 2.2 Auxiliary commands

Besides the commands for carrying out deductions, LCFsmall has the following commands:

SHOW LINE L1;

SHOW LINE L1: L2;

In the first case the step L1 is printed. In the second case all the steps between L1 and L2 are printed.

FETCH *FILENAME*;

All the LCFsmall commands contained in the file *FILENAME* are executed. Each command is treated exactly as if typed at the console. So the user may prepare all the commands on a file and then generate a proof by fetching this file.

CANCEL;

CANCEL L1;

In the first case the last step in the proof is deleted. In the second case all the steps from the last one to L1 (included) are deleted. If L1 is less or equal to one, the entire proof is cancelled!

Section 2.3 Messages from LCFsmall

The following list includes all the messages printed by LCFsmall:

SYNTAX ERROR; TRY AGAIN

This is printed whenever a command is improperly typed.

NASTY *COMMAND*

This error message is printed by any command whenever it cannot be executed because some condition isn't satisfied. For instance, if you are trying to FIXP a nonexisting step or a step whose first awff is not a fixpoint definition you will get NASTY FIXP.

THE LAST LINE IN THE PROOF IS N

YOU HAVE DEMOLISHED YOUR PROOF

One of the above sentences is the answer of the system after executing a cancel command.

You may also obtain something like

3246 ILL MEM REF FROM ATOM

if you have messed up something with LISP! However this shoudn't happen.

Section 2.4 How to use LCFsmall

If you want to prove something use LCF! Anyway, if you really want to use LCFsmall type:

```
R LCFSML
```

you are at LISP level and you will get a star. If you type

```
(INIT)
```

you 'll get some stars and then you are ready to prove. To stop a proof type

```
$
```

You'll receive the message END OF PROOF. Now you are again at LISP level. Typing

```
(RESUME)
```

will make you to go on with the old proof. If you want to start a new proof, type

```
(INIT)
```

Your core image may be saved for later use by the command

```
↑C
SAVE FILENAME
```

Section 2.5 Examples of proofs

Two sample LCFsmall proofs are given here. They concerns the CASE and INDUCT commands. The corresponding LCF proofs are very different. In fact, they are done using the subgoaling mechanism.

The first statement we have proved is the following property of conditional expressions:

```
(P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)
```

All the commands have been typed in the file TSTCS. They are:

```
CONDT (TT→(P(X)→C1,C2),(P(X)→C1,C2));
CONDU (UU→(P(X)→C1,C2),(P(X)→C1,C2));
CONDU (UU→C1,C2);
CONDF (FF→(P(X)→C1,C2),(P(X)→C1,C2));
SYM 3;
SUBST 5 OCC 2 IN 2;
```

```
ASSUME P(X)≡TT;
ASSUME P(X)≡UU;
ASSUME P(X)≡FF;
SYM 7;
SYM 8;
SYM 9;
SUBST 10 OCC 1 IN 1;
SUBST 11 OCC 1 IN 6;
SUBST 11 OCC 1 IN 14;
SUBST 12 OCC 1 IN 4;
CASES 13 15 16 P(X);
```

The file is then fetched and the proof is done. The printout of LCFsmall is

```
R LCFSML
(INIT)
FETCH TSTCS;


****1  (TT→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)
****2  (UU→(P(X)→C1,C2),(P(X)→C1,C2))≡UU
****3  (UU→C1,C2)≡UU
****4  (FF→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)
****5  UU≡(UU→C1,C2)
****6  (UU→(P(X)→C1,C2),(P(X)→C1,C2))≡(UU→C1,C2)
****7  P(X)≡TT      (7)
****8  P(X)≡UU      (8)
****9  P(X)≡FF      (9)
****10  TT≡P(X)     (7)
****11  UU≡P(X)     (8)
****12  FF≡P(X)     (9)
****13  (P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)     (7)
****14  (P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(UU→C1,C2)     (8)
****15  (P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)     (8)
****16  (P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)     (9)
****17  (P(X)→(P(X)→C1,C2),(P(X)→C1,C2))≡(P(X)→C1,C2)
****
*****$

END OF PROOF
NIL
*↑C
↑C
```

The next example is taken from Milner 1972, section 3.1. The statement to be proved is:

F⊂G ASSUME F≡[αF.FUN(F)], G≡FUN(G).

The commands, typed in the file TSTIND are:

```
ASSUME F≡[αF.FUN(F)];
ASSUME G≡FUN(G);
ASSUME F1⊂G;
```

```
MINI G;
APPL FUN 3;
SYM 2;
SUBST 6 OCC 1 IN 5;
INDUCT 1 4 3 7 F1;
```

The printout of LCFsmall is:

```
R LCFSML
(INIT)
FETCH TSTIND;


****1  F≡[αF.FUN(F)]     (1)
****2  G≡FUN(G)    (2)
****3  F1⊂G     (3)
****4  UU⊂G
****5  FUN(F1)⊂FUN(G)     (3)
****6  FUN(G)≡G     (2)
****7  FUN(F1)⊂G     (2 3)
****8  F⊂G     (1 2)
****
*****$

END OF PROOF
NIL
*↑C
↑C
```

The length of the two above LCFsmall proofs is comparable with that of their corresponding LCF proofs. However, as soon as the proof becomes more complex and a considerable amount of substitutions and conversions have to be done, the subgoaling mechanism and -more important- the simplification algorithm of LCF become vital.

SECTION 3 Description of the program

The MLISP2 program for LCFsmall is completely listed in the appendices 1 through 7. In the following sections, the various components of the program are described. They are:

1) parser
2) top level *driver*
3) printing routines
4) commands
5) auxiliary functions
6) functions manipulating the data structure

Section 3.1 The Parser

3.1.1 Scanning primitives

This code implements a backupable scanner. It uses an array, TSTACK, to store "tokens" as they are scanned. Actually the scanner returns both a type and a value, where "value" is the atom scanned and "type" is:

IDENT if the value is an identifier
NUMBER if the value is a number
DEL if the value is a delimiter.

Two global variables are used to keep track of what token we are looking at in the input stream. They are PC and ENDSTACK. PC points into TSTACK at the place the LCFsmall scanner is looking. ENDSTACK is the last location in TSTACK that has been filled from the current input. TSTACK is necessary because scan destroys the input stream, and the LCFsmall parser, being top down, needs to back up over the input. The main accessing routine for TSTACK is the function **tstack** which calls **scan** if not enough tokens have been read.

scan(): returns a pair consisting of the token scanned and its type.

setup(): sets PC=0 and ENDSTACK←0 and declares the array TSTACK.

token: simply advances the LCFsmall scanner.

tokenv(): advances the scanner and returns the value of the new thing pointed to.

tokent(): advances the scanner and returns the type of the new thing pointed to.

tstack(n): finds the n-th element of TSTACK, if its not there it calls **scan** until it is.

peekv(n): returns the n-th token ahead of PC.

peekt(n): returns the type of the n-th token ahead of PC.

flush(): starts the LCFsmall scanner over by setting PC=0 and ENDSTACK=0.

nextv(x): returns T if the value of the next token is x, NIL otherwise.

nextt(x): returns T if the type of the next token is x, NIL otherwise.

The function **scan** was not written with efficiency in mind. It uses ordinary LISP functions whose properties we know about. This is because we hope someday to prove the correctness of this program. Note that the only functions not definable in pure LISP are READLIST, ASCII, TYI, and TSTACK. Arrays could easily be eliminated in favor of lists. The array TYPE stores the type of a character, 0 for letters, 1 for digits, 2 for delimiters, 3 for characters to be ignored when building tokens (like form feeds). The special global variables can be eliminated from the code in favor of pure LISP in the standard way.

3.1.2 The wff parser

Rather than describing everything in detail we will explain the parser by explaining some examples. Consider

```
EXPR TERM();
        BEGIN NEW START,REP,X,Y;START←PC;
        IF X←SIMPLTERM() THEN REP←X ELSE RETURN NIL;
A;       START←PC;
        IF LPAR()∧(Y←TERM())∧RPAR() THEN REP←('?!APPLY CONS REP CONS Y) ALSO GO A;
        PC←START;
        RETURN(REP);END;
```

The local variable **START** is to remember where the global variable **PC** was pointing when the function was entered, i.e. **START←PC**. The convention for a parsing function is that either it exits successfully with a non NIL value and leaves **PC** pointing to the next token to be looked at or it returns NIL and leaves the value of **PC** as it was when the function was entered. The code

```
        IF X←SIMPLTERM() THEN REP←X ELSE RETURN NIL;
```

checks if a **SIMPLTERM** is scanned. In this case **REP** gets it as a value. If not (by our convention) **SIMPLTERM** returns NIL, and **PC** is left as it was, so **TERM** returns NIL and **PC** remains unchanged. If we have found a **SIMPLTERM**, **TERM** has succeeded and we enter a loop, update the place in the input stream we backup to when we exit **TERM** and look for repetitions of a left parenthesis (**LPAR**), followed by e **TERM**, followed by a right parenthesis (**RPAR**).

```
A;      START←PC;
        IF LPAR()∧(Y←TERM())∧RPAR() THEN REP←('?!APPLY CONS REP CONS Y) ALSO GO A;
```

After each successful repetition **REP** gets the internal representation of an application term, i.e. **F(x)→(APPLY! F x)**. When the loop test eventually fails we restore **PC** and return the term stored in **REP**.

Section 3.2 Top level driver

LCFsmall is started by the INIT function. This and the other top level functions are listed in appendix 2. INIT sets the base for numbers to 10, initializes the scanner and then initializes the proof. PROOF, the global variable which keeps record of the proof, is set to NIL and PFLENGTH, the proof length, is set to 0. Then RESUME is called. It takes into account the fact that the input commands may be read from the console or from a fetched file. It calls the function LCFPROOF which builds up the proof by a *read execute-write* loop.

LCFPROOF makes a test on the content of the input buffer. If its first character is $, then an end of proof message is typed and the proof is stopped. If a command is parsed and executed the loop goes on. The function LINE controls the execution of LCF commands. After a command has been successfully parsed and executed, if the value returned is a proof step, then it is added to the proof.

If none of the expected command is parsed, the input buffer is scanned by the function BADLINE until the first semicolon is met. Then an error message is printed.

Section 3.3 Printing routines

The printing routines are listed in appendix 3. They depend on the internal representation of terms, awffs, wffs and proof steps, which is described in section 3.6.

PRINTAWFF is the printing routine for terms and awffs. They are transformed from the internal prefix form to a parenthetized form.

PRINTMES prints messages. it takes the string to be printed as argument. PRINTM is used to print a message when some steps in the proof have been cancelled. The string to be written is fixed, the argument of PRINTM is the proof-length after the cancellation.

PRINTNEWLINE prints the newly generated line, whenever a command is successfully executed. The stepnumber, the wff and its dependencies are printed. PRINTLINE is like PRINTNEWLINE, but it may print any step in the proof, not necessarily the last one. It prints also the reason of the step.

PRINTLST is an auxiliary printing routine which prints a list of awffs separated by blanks.

Section 3.4 Commands

The commands are shown in appendices 4 and 5. They are listed in the same order as they are describd in sections 2.1 and 2.2. Every command is realized by two functions. The first one performs a check on the syntax of the input sentence. If the expected command is successfully parsed then the corresponding semantic function is called, otherwise the pointer is restarted in the input buffer. This allows the input sentence to be tested again to see if we are faced with another command or if there is a syntax error in the input. Each semantic function performs a series of tests to see whether or not the conditions for the applicability of the corresponding rule are met. In this case it returns a new step to be added to the proof, otherwise it returns the message NASTY *COMMAND*.

We think that all the syntactic and semantic functions realizing the LCFsmall commands are sufficiently clear, after having read the description of the commands given in sections 2.1 and 2.2.

Section 3.5 Auxiliary functions

The auxiliary functions and predicates used in defining the commands are listed in appendices 6 and 7. Appendix 7 contains the predicates and functions directly dealing with the data structure, they will be described in the next section. The functions and predicates listed in appendix 6 have been divided into three groups and will be discussed in the three following subsections.

3.5.1 Predicates on free and bound occurrences of variables

NOTBNDVT(V,TRM) is a predicate true if V has no bound occurrences in TRM. BOUNDV is its negation.

NOTFRVT(V,TRM) is a predicate true if V has no free occurrences in TRM. FREEV is its negation.

NOTFREVW(V,WF) is true if V has no free occurrences in the wff WF. NOTFREE(V,LN) is true if V doesn't occur free in the wffs associated with the stepnumbers in the list LN.

ISFREEFORT(X,V,TRM) is true if X (a term or a variable) may be substituted for V in the term TRM without conflicts of bound variables. ISFREEFORW(X,V,WF) is the analogue for wffs.

3.5.2 Functions used in INCL, CUT, CASES, SHOW

The functions described in this section are listed in appendix 6.2.

PICKUP is used in the command INCL for selecting the n-th awff in a wff.

INCLTEST(LN,WF) uses TESTM. It is used in CUT to check if every wff associated with the stepnumbers in the list LN appears in WF.

TESTCASES and TESTC are used in testing the applicability of the cases rule. FIND and REMOVE are used in building up the dependency part of the step generated by the CASES command.

OPT is used in the SHOW command to parse an optional part in the input string.

3.5.3 Conversion and substitution routines

The conversion and substitution routines are listed in appendix 6.3.

CONVT(TRM) performs all the possible lambda-conversions on TRM. If it is an identifier, no conversion can be done. If it is composed of various parts, then the conversion is recursively done on them. If it is an application term, then tests are performed to see if a conversion can be done and if the resulting term can be further converted.

SUBSTG(TRM,X,V1) is the "general" substitution routine. X, a variable or a term, replaces V1 in all its free occurrences in TRM. A test is done on TRM and X is recursively substituted in all the components of TRM. When faced with a lambda-term or a mu-term a test is done to detect conflicts of variables.

ACONV(TRM,V1,V2) performs an alpha-conversion on TRM. V1 replaces V2 in its first bound nonconflicting occurrence.

SUBW(AWF1,AWF2,N) is an auxiliary function used in the command SUBST, when it is applied to two stepnumbers. AWF1 is the awff in which the substitution takes place. The term at the left hand side of AWF2, denoted as TRM1, replaces the term at the right hand side of AWF2, denoted as TRM2, in its N-th occurrence. The global variable SUBCOUNT is set to N, it will mark the occurrence where the substitution must be done. The substitution is first attempted on the term at the left hand side of AWF1. If not performed there, then it is attempted in the term at the right hand side of AWF1.

SUBSTTT(TRM1,TRM2,TRM3,N) is used by the command SUBST when its last argument is a term. TRM2 replaces TRM3 in its N-th occurrence in TRM1.

DOSUBST(TRM1,TRM2,TRM3) is the auxiliary function that performs the substitution of TRM2 for TRM1 in TRM1. A test is done on TRM1 and the substitution is recursively attempted on its various parts. SUBCOUNT is decremented whenever an occurrence is found and, when its value is 0 the substitution takes place. Occurrences where conflicts arise among variables are not counted.

Section 3.6 The Data Structure

All the functions directly manipulating the data structure are listed in appendix 7.

In appendix 7.1 all the *constructors* are listed. By constructor we mean a function that assembles structured data.

MKCONDTERM, MKAPPLTERM, MKLAMBDATERM and MKMUTERM define the internal representation of terms. They are represented as LISP S-expressions whose first element denotes the nature of the term and is followed by the components of the term. Awffs are assembled by MKAWFF. They are S-expressions whose first element is the relation symbol ≡ or ⊂. MKWFF assembles wffs of just one awff. In general wffs may be lists of more than one awff. For instance those produced by the function UNIONW (see appendix 7.4) used in the command CONJ.

The proof is represented as a list, initially it is set to NIL. Each step is added to this list by the function ADDLINE (see appendix 7.4) and is assembled by the constructor MKPROOFSTEP. Proof steps have the form of a list of three elements: a wff, a list of dependencies and a reason assembled by the constructor REASON. The function ADDLINE puts the stepnumber in front of each proof step.

Appendix 7.2 contains the list of all the *selectors* used in retrieving the various components of the terms, awffs and the proof.

Appendix 7.3 contains a list of predicates used in the program. These predicates are tests on the nature of terms, awffs etc.

Some miscellaneous functions are listed in appendix 7.1: UNIONOF is the set theoretic union for lists of numnbers, UNIONW is the set theoretic union for wffs, manely for lists of awffs. ADDLINE (see above) increments the variable PFLENGTH (proof length) by 1 and adds a new step to the proof. SEARCH is used to search steps in the proof, LNT gives the length of a list, and finally SUBWV(WF,X,V) substitutes X for each occurrence of V in WF. It is used in the command INDUCT.

REFERENCES

Hoare, C.A.R.,
1973 *Recursive Data Structure*
Artificial Intelligence Memo No. 223, Stanford University (1973).

Milner, R.,
1972 *Logic for computable functions, description of a machine implementation*
Artificial Intelligence Memo No. 169, Stanford University (1972).

Newey, M.,
1974 *Formal Semantics of LISP with Applications to Program Correctness*
Forthcoming Ph. D. Dissertation, Stanford University, 1974.

Smith, D.C. and Enea H.J.,
1973 *MLISP2*
Artificial Intelligence Memo No. 195, Stanford University (1973).

Weyhrauch, R.W. and Milner, R.,
1972 *Program Semantics and Correctness in a Mechanized Logic,*
Proc. 1st USA-Japan Computer Conf., Tokyo (1972).

APPENDIX 1

THE PARSER

1.1 Special variables

```
        PC,
        ENDSTACK,
        PROOF,
        PFLENGTH,
        SUBCOUNT;
```

1.2 Scanner for LCFsmall

```
EXPR readlist(X);
        READLIST(ASCII(OCTAL 57) CONS X);

EXPR scan( :X);
        IF EQ(X←TYPE(CHAR),0) THEN idscan()
        ELSE IF EQ(X,1) THEN numscan()
        ELSE IF EQ(X,2) THEN delscan()
        ELSE CHAR←TYI() ALSO scan();

EXPR idscan();
        BEGIN NEW TOKEN,X;
        TOKEN←<ASCII(CHAR)>;
A;       IF EQ(X←TYPE(CHAR←TYI()),0)vEQ(X,1)
          THEN TOKEN←ASCII(CHAR) CONS TOKEN ALSO GO A;
        RETURN(readlist(REVERSE(TOKEN)) CONS 'IDENT); END;

EXPR numscan();
        BEGIN NEW TOKEN;
        TOKEN←<ASCII(CHAR)>;
A;       IF EQ(TYPE(CHAR←TYI()),1)
          THEN TOKEN←ASCII(CHAR) CONS TOKEN ALSO GO A;
        RETURN(readlist(REVERSE(TOKEN)) CONS 'NUMBER); END;

EXPR delscan();
        BEGIN NEW TOKEN;
        TOKEN←<ASCII(CHAR)>;
        CHAR←TYI();
        RETURN(readlist(TOKEN) CONS 'DEL);END;

EXPR setup();
        BEGIN NEW X;
        ARRAY(TYPE,36,CONS(0,127));
        ARRAY(TSTACK,T,CONS(0,500));
        FOR X←0 TO 127 DO TYPE(X)←2;
        FOR X←OCTAL 011 TO OCTAL 015 DO TYPE(X)←3;
        FOR X←OCTAL 060 TO OCTAL 071 DO TYPE(X)←1;
        FOR X←OCTAL 101 TO OCTAL 132 DO TYPE(X)←0;
        FOR X←OCTAL 141 TO OCTAL 172 DO TYPE(X)←0;
        TYPE(OCTAL 040)←3; TYPE(OCTAL 175)←3; TYPE(OCTAL 177)←3; END;
```

1.3 Parsing primitives

```
EXPR token(); PC←PC+1;

EXPR tokenv(); CAR tstack(PC←PC+1);
EXPR tokent(); CDR tstack(PC←PC+1);

EXPR tstack(N);
       IF ENDSTACK LESSP N
        THEN FOR NEW I←(ENDSTACK+1) TO N DO TSTACK(I)←scan()
             ALSO ENDSTACK←N
             ALSO TSTACK(N)
        ELSE TSTACK(N);

EXPR peekv(N); CAR tstack(PC+N);
EXPR peekt(N); CDR tstack(PC+N);

EXPR flush(); BEGIN PC←0; ENDSTACK←0;END;

EXPR nextv(X); EQ(X,CAR tstack(PC+1));
EXPR nextt(X); EQ(X,CDR tstack(PC+1));
```

1.4 Parser

```
EXPR TERM();
        BEGIN NEW START,REP,X,Y;START←PC;
        IF X←SIMPLTERM() THEN REP←X ELSE RETURN NIL;
A;       START←PC;
        IF LPAR()∧(Y←TERM())∧RPAR()
         THEN REP←('?!APPLY CONS REP CONS Y) ALSO GO A;
        PC←START;
        RETURN(REP);END;

EXPR CONDTERM();
        BEGIN NEW START,X,Y,Z; START←PC;
```

```
       IF LPAR()∧(X←TERM())∧RARROW()∧(Y←TERM())∧COMMA()∧(Z←TERM())∧RPAR()
        THEN RETURN( '?!COND CONS X CONS Y CONS Z);
       PC←START;END;

EXPR LAMBDATERM();
       BEGIN NEW START,X,Y; START←PC;
       IF LSQBRACKET()∧lambda()∧(X←IDENT())∧PERIOD()∧(Y←TERM())∧RSQBRACKET()
        THEN RETURN( '?!LAMBDA CONS X CONS Y);
       PC←START;END;

EXPR MUTERM();
       BEGIN NEW START,X,Y; START←PC;
       IF LSQBRACKET()∧MU()∧(X←IDENT())∧PERIOD()∧(Y←TERM())∧RSQBRACKET()
        THEN RETURN( '?!MU CONS X CONS Y);
       PC←START;END;

EXPR SIMPLTERM();
       BEGIN NEW START,X;START←PC,
       IF (X←IDENT())    ∨
          (X←CONDTERM()) ∨
          (X←LAMBDATERM()) ∨
          (X←MUTERM()) ∨
          (LPAR()∧(X←TERM())∧RPAR())
        THEN RETURN X;
       PC←START;END;

EXPR AWFF();
       BEGIN NEW START,X,R,Y; START←PC;
       IF (X←TERM())∧(R←REL())∧(Y←TERM())
        THEN RETURN( R CONS X CONS Y);
       PC←START;END;

EXPR WFF();
       BEGIN NEW START,REP,X;START←PC;
       IF X←AWFF() THEN REP←<X> ELSE RETURN NIL;
A;      START←PC;
       IF COMMA()∧(X←AWFF()) THEN REP←<X>@REP ALSO GO A;
       PC←START;
       RETURN(REP);END;


EXPR IDENT(); IF EQ(peekt(1),'IDENT) THEN tokenv() ELSE NIL;
EXPR NUMBER(); IF EQ(peekt(1),'NUMBER) THEN VALUE(tokenv()) ELSE NIL;
EXPR REL(); IF nextv('?≡)∨nextv('?⊂) THEN tokenv() ELSE NIL;
EXPR CHECK(X);     IF nextv(X) THEN token() ELSE NIL;
EXPR SC();        IF nextv('?;) THEN token() ELSE NIL;
EXPR LPAR();       IF nextv('?() THEN token() ELSE NIL;
EXPR RPAR();       IF nextv('?)) THEN token() ELSE NIL;
EXPR RARROW();      IF nextv('?→) THEN token() ELSE NIL;
EXPR COMMA();       IF nextv('?,) THEN token() ELSE NIL;
EXPR COLON();      IF nextv('?:) THEN token() ELSE NIL;
EXPR DOLLAR();      IF nextv('?$) THEN token() ELSE NIL;
EXPR PERIOD();     IF nextv('?.) THEN token() ELSE NIL;
EXPR LSQBRACKET(); IF nextv('?[) THEN token() ELSE NIL;
EXPR RSQBRACKET(); IF nextv('?]) THEN token() ELSE NIL;
EXPR lambda();     IF nextv('?λ) THEN token() ELSE NIL;
EXPR MU();         IF nextv('?α) THEN token() ELSE NIL;


EXPR VALUE(X);
     (READLIST(CDR(EXPLODE X)));
```

APPENDIX 2

TOP LEVEL ROUTINES

```
EXPR INIT();
     BEGIN
     LISPINIT();
     SCNINIT();
     LCFINIT();
     END;

EXPR LISPINIT();
     BEGIN
     ?*NOPOINT←T;
     BASE ←10.;
     IBASE ←10.;
     END;

EXPR SCNINIT();
     BEGIN
     CHAR ← 40;
     PC←1;
     ENDSTACK←0;
     setup();
     END;

EXPR LCFINIT();
     BEGIN
     PROOF←NIL;
     PFLENGTH ← 0;
     RESUME();
     END;

EXPR RESUME();
     BEGIN NEW X;
A;   X←ERRSET(LCFPROOF());
     IF EQ(X,'?$EOF?$) THEN INC(NIL,T) ALSO flush() ALSO GO A;
     END;

EXPR LCFPROOF();
      BEGIN
   A; PRINC(TERPRI("****"));
      IF DOLLAR() THEN PRINTMES("END OF PROOF")
                ALSO flush()
                ALSO RETURN(PRINC("  "));
      IF LINE() v BADLINE() THEN flush() ALSO GO A;
      END;

EXPR LINE();
     BEGIN NEW NC;
     IF (NC←FETCH()) v (NC←SHOW()) v (NC←CANCEL()) THEN RETURN(NC);
     IF (NC←ASSUME()) v (NC←INCL()) v
        (NC←REFL1()) v (NC←REFL2()) v
      (NC←MIN1()) v (NC←MIN2()) v
      (NC←ALPHACONV()) v(NC←SUBST()) v
      (NC←ABSTR()) v (NC←FIXP()) v
      (NC←CONDT()) v (NC←CONDF()) v
      (NC←CONDU()) v (NC←EQUIV()) v
      (NC←HALF()) v (NC←SYM()) v
      (NC←TRANS()) v (NC←APPL()) v
      (NC←CONJ()) v (NC←CUT()) v
      (NC←CASES()) v (NC←INDUCT()) v
      (NC← CONV()) v (NC←ETACONV())
     THEN (IF ISLINE(NC) THEN  ADDLINE(NC) ALSO PRINTNEWLINE());
    RETURN (NC);
    END;

EXPR BADLINE();
    BEGIN
  A; IF ¬nextv('?;) THEN token() ALSO GO A;
    PRINTMES( "SYNTAX ERROR;TRY AGAIN");
    RETURN (PRINC("  "));
    END;
```

APPENDIX 3

PRINTING ROUTINES

```
EXPR PRINTAWFF(AWF);
        BEGIN NEW CR;
        IF ATOM(AWF) THEN RETURN PRINC(AWF);
        CR←CAR(AWF);
        IF EQ(CR,'?≡) ∨ EQ(CR,'?⊂)
         THEN BEGIN PRINTAWFF(CADR AWF);
                    PRINC(CR);
                    PRINTAWFF(CDDR AWF); END;
        IF EQ(CR,'?!APPLY)
         THEN BEGIN PRINTAWFF(CADR AWF);
                    PRINC('?( );
                    PRINTAWFF(CDDR AWF);
                    PRINC('?));        END;
        IF EQ(CR,'?!COND)
         THEN BEGIN PRINC('?( );
                    PRINTAWFF(CADR AWF);
                    PRINC('?→);
                    PRINTAWFF(CADDR AWF);
                    PRINC('?,);
                    PRINTAWFF(CDDDR AWF);
                    PRINC('?));        END;
        IF EQ(CR,'?!LAMBDA)
         THEN BEGIN PRINC('?[?λ);
                    PRINTAWFF(CADR AWF);
                    PRINC('?.);
                    PRINTAWFF(CDDR AWF);
                    PRINC('?]);        END;
        IF EQ(CR,'?!MU)
         THEN BEGIN PRINC('?[?α);
                    PRINTAWFF(CADR AWF);
                    PRINC('?.);
                    PRINTAWFF(CDDR AWF);
                    PRINC('?]);        END;
        END;

EXPR PRINTMES(X);
     TERPRI(PRINC(TERPRI(X)));

EXPR PRINTM(N);
     BEGIN
     PRINC(TERPRI("THE LAST LINE IN THE PROOF IS: "));
     RETURN(TERPRI(PRINC(N)));
     END;

EXPR PRINTNEWLINE();
     BEGIN NEW X;
     X←PROOF[1];
     PRINC(X[1]); IF (X[1]≥10) THEN PRINC("  ") ELSE PRINC("   ");
     PRINTLST(X[2]); PRINC("      ");
     RETURN PRINC(IF NULL(X[3]) THEN " " ELSE X[3]); END;
```

```
EXPR PRINTLINE(X);
    BEGIN
    PRINC(X[1]); IF (X[1]≥10) THEN PRINC("  ") ELSE PRINC("   ");
    PRINTLST(X[2]); PRINC("      ");
    PRINC(IF NULL(X[3]) THEN " " ELSE X[3]); PRINC("       ");
    IF ATOM(X[4]) THEN RETURN PRINC(X[4]) ELSE  RETURN PRINTLST(X[4]);
    END;

EXPR PRINTLST(X);
    IF NULL(CDR X) THEN PRINTAWFF(X[1]) ELSE
    BEGIN  PRINTAWFF(X[1]);
           PRINC(" ");
           RETURN PRINTLST(CDR X);END;
```

APPENDIX 4

INFERENCE COMMANDS

```
EXPR ASSUME();
    BEGIN NEW AWF,START; START←PC;
    IF CHECK('ASSUME) ∧ (AWF←AWFF()) ∧ SC()
     THEN RETURN ASSUMESEM(AWF); PC←START;
    END;

EXPR ASSUMESEM(AWF);
    MKPROOFSTEP(<AWF>,<PFLENGTH + 1>,'ASSUME );

EXPR INCL();
    BEGIN NEW L1,N,START; START←PC;
    IF CHECK('INCL) ∧ (L1←NUMBER()) ∧ (N←NUMBER()) ∧ SC()
     THEN RETURN INCLSEM(L1,N); PC←START;
    END;

EXPR INCLSEM(L1,N :WF);
    IF ISPROOFSTEP(L1) ∧ISINCL(N,WF←WFFOF(L1))
      THEN MKPROOFSTEP(PICKUP(WF,N),DEPOF(L1),REASON('INCL,<L1,N>))
         ELSE PRINTMES("NASTY INCL");

EXPR CONJ();
    BEGIN NEW L1,L2,START; START←PC;
    IF CHECK('CONJ) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧ SC()
     THEN RETURN CONJSEM(L1,L2); PC←START;
    END;

EXPR CONJSEM(L1,L2);
    IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP( L2)
     THEN MKPROOFSTEP(UNIONW(WFFOF(L1),WFFOF(L2)),
                        UNIONOF(DEPOF(L1),DEPOF(L2)),
                        REASON('CONJ,<L1,L2>))
     ELSE PRINTMES("NASTY CONJ");

EXPR CUT();
    BEGIN NEW L1,L2,START;START←PC;
    IF CHECK('CUT) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧ SC()
     THEN RETURN CUTSEM(L1,L2); PC←START;
    END;

EXPR CUTSEM(L1,L2);
    IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2) ∧ INCLTEST(DEPOF(L2),WFFOF(L1))
       THEN MKPROOFSTEP(WFFOF(L2),DEPOF(L1),REASON('CUT,<L1,L2>))
       ELSE PRINTMES("NASTY CUT");

EXPR HALF();
    BEGIN NEW L1,START; START←PC;
    IF CHECK('HALF) ∧ (L1←NUMBER()) ∧ SC()
     THEN RETURN HALFSEM(L1); PC←START;
    END;
```

```
EXPR HALFSEM(L1 :AWF);
    IF ISPROOFSTEP(L1) ∧ ISEQUIVAWFF(AWF←AWFFOF(L1))
       THEN MKPROOFSTEP(MKWFF('?⊂,FSTERMOF(AWF),SNTERMOF(AWF)),DEPOF(L1),
                             REASON('HALF,<L1>))
       ELSE PRINTMES("NASTY HALF");

EXPR SYM();
    BEGIN NEW L1,START; START←PC;
    IF CHECK('SYM) ∧ (L1←NUMBER()) ∧ SC()
     THEN RETURN SYMSEM(L1); PC←START;
    END;

EXPR SYMSEM(L1 :AWF);
    IF ISPROOFSTEP(L1) ∧ ISEQUIVAWFF(AWF←AWFFOF(L1))
       THEN MKPROOFSTEP(MKWFF('?≡,SNTERMOF(AWF),FSTERMOF(AWF)),DEPOF(L1),
                  REASON('SYM,<L1>))
       ELSE PRINTMES("NASTY SYM");

EXPR TRANS();
    BEGIN NEW L1,L2,START;START←PC;
    IF CHECK('TRANS) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧ SC()
     THEN RETURN TRANSSEM(L1,L2); PC←START;
    END;

EXPR TRANSSEM(L1,L2 :AWF1,AWF2,REL);
    IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2)
      ∧ EQUAL(SNTERMOF(AWF1←AWFFOF(L1)),FSTERMOF(AWF2←AWFFOF(L2)))
    THEN (IF ISEQUIVAWFF(AWF1) ∧ ISEQUIVAWFF(AWF2)
               THEN REL ← ('?≡) ELSE REL ← ('?⊂))
    ALSO  MKPROOFSTEP(MKWFF(REL,FSTERMOF(AWF1),SNTERMOF(AWF2)),
                       UNIONOF(DEPOF(L1),DEPOF(L2)),
                       REASON('TRANS,<L1,L2>))
    ELSE PRINTMES("NASTY TRANS");

EXPR APPL();
    BEGIN NEW L1,TRM,START; START←PC;
    IF CHECK('APPL) ∧ (TRM←TERM()) ∧ (L1←NUMBER()) ∧ SC()
        THEN RETURN APPLSEM1(TRM,L1); PC←START;
    IF CHECK('APPL) ∧ (L1←NUMBER()) ∧ (TRM←TERM()) ∧ SC()
         THEN RETURN APPLSEM2(L1,TRM); PC←START;
    END;

EXPR APPLSEM1(TRM,L1:AWF);
    IF ISPROOFSTEP(L1)  THEN
     MKPROOFSTEP(MKWFF(RELOF(AWF←AWFFOF(L1)),MKAPPLTERM(TRM,FSTERMOF(AWF)),
                       MKAPPLTERM(TRM,SNTERMOF(AWF))),
                          DEPOF(L1),REASON('APPL,<TRM,L1>))
     ELSE PRINTMES("NASTY APPL");

EXPR APPLSEM2(L1,TRM:AWF);
    IF ISPROOFSTEP(L1) THEN
     MKPROOFSTEP(MKWFF(RELOF(AWF←AWFFOF(L1)),MKAPPLTERM(FSTERMOF(AWF),TRM),
                        MKAPPLTERM(SNTERMOF(AWF),TRM)),
                          DEPOF(L1),REASON('APPL,<L1,TRM>))
    ELSE PRINTMES("NASTY APPL");
```

```
EXPR ABSTR();
    BEGIN NEW L1,V1,START;START←PC;
    IF CHECK('ABSTR) ∧ (L1←NUMBER()) ∧ (V1←IDENT()) ∧ SC()
    THEN RETURN ABSTRSEM(L1,V1); PC←START;
    END;

EXPR ABSTRSEM(L1,V1 :AWF);
     BEGIN
     IF ISPROOFSTEP(L1) ∧  NOTFREE(V1,DEPOF(L1)) THEN
          AWF←AWFFOF(L1) ALSO RETURN(MKPROOFSTEP(MKWFF(RELOF(AWF),
                                    MKLAMBDATERM(V1,FSTERMOF(AWF)),
                                    MKLAMBDATERM(V1,SNTERMOF(AWF))),
                               DEPOF(L1),REASON('ABSTR,<L1,V1>)))
       ELSE RETURN(PRINTMES("NASTY ABSTR")); END;

EXPR CASES();
    BEGIN NEW L1,L2,L3,TRM,START; START←PC;
    IF CHECK('CASES) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧
        (L3←NUMBER()) ∧ (TRM←TERM()) ∧ SC()
     THEN RETURN CASESSEM(L1,L2,L3,TRM); PC←START;
    END;

EXPR CASESSEM(L1,L2,L3,TRM:WF1,WF2,D1,D2,D3);
    IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2) ∧ ISPROOFSTEP(L3) ∧
         EQUAL(WF1←WFFOF(L1),WF2←WFFOF(L2)) ∧
         EQUAL(WF2,WFFOF(L3)) ∧
             TESTCASES(D1←DEPOF(L1),D2←DEPOF(L2),D3←DEPOF(L3),TRM)
           THEN MKPROOFSTEP(WF1,UNIONOF(REMOVE(D1,FIND(D1,TRM,'TT)),
                            UNIONOF(REMOVE(D2,FIND(D2,TRM,'UU)),
                                   REMOVE(D3,FIND(D3,TRM,'FF)))),
                                  REASON('CASES,<L1,L2,L3,TRM>))
           ELSE PRINTMES("NASTY CASES");

EXPR INDUCT();
    BEGIN NEW L1,L2,L3,L4,V1,START; START←PC;
    IF CHECK('INDUCT) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧ (L3←NUMBER()) ∧
          (L4←NUMBER()) ∧ (V1←IDENT()) ∧ SC()
     THEN RETURN INDUCTSEM(L1,L2,L3,L4,V1); PC←START;
    END;

EXPR INDUCTSEM(L1,L2,L3,L4,V1);
    BEGIN NEW AWF1,WF3,FIX,MT,BV,MAT,FUNV1;
    IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2) ∧ ISPROOFSTEP(L3) ∧ISPROOFSTEP(L4) ∧
        ISMUTERM(MT←SNTERMOF(AWF1←AWFFOF(L1))) ∧
       ISFREEFORT(FIX←FSTERMOF(MT),BV←BVAROF(MT),MAT←MATRIXOF(MT)) ∧
       ISFREEFORW('UU,V1,WF3←WFFOF(L3)) ∧
       ISFREEFORT(V1,BV,MAT) ∧
       ISFREEFORW(FUNV1←SUBSTG(MAT,V1,BV),V1,WF3) ∧
       ISFREEFORW(FIX,V1,WF3) ∧
       EQUAL(WFFOF(L2),SUBWV(WF3,'UU,V1)) ∧
       EQUAL(WFFOF(L4),SUBWV(WF3,FUNV1,V1)) ∧
              MEMQ(L3,DEPOF(L4))
    THEN RETURN MKPROOFSTEP(SUBWV(WF3,FSTERMOF(AWF1),V1),
                  UNIONOF(UNIONOF(DEPOF(L1),DEPOF(L2)),
                    REMOVE(UNIONOF(DEPOF(L3),DEPOF(L4)),L3)),
                        REASON('INDUCT, <L1,L2,L3,L4,V1>))
     ELSE  PRINTMES("NASTY INDUCT");
     END;

EXPR CONV();
     BEGIN NEW L1,TRM,START;START←PC;
     IF CHECK('CONV) ∧ (L1←NUMBER()) ∧ SC()
        THEN RETURN CONVSEM1(L1); PC←START;
     IF CHECK('CONV) ∧ (TRM←TERM()) ∧ SC()
        THEN RETURN CONVSEM2(TRM); PC←START;
     END;

EXPR CONVSEM1(L1:AWF);
     IF ISPROOFSTEP(L1)
      THEN MKPROOFSTEP(MKWFF(RELOF(AWF←AWFFOF(L1)),
               CONVT(FSTERMOF(AWF)),CONVT(SNTERMOF(AWF))),
                 DEPOF(L1),REASON('CONV,<L1>))
      ELSE PRINTMES("NASTY CONV");

EXPR CONVSEM2(TRM);
     MKPROOFSTEP(MKWFF('?≡,TRM,CONVT(TRM)),'NODEP,REASON('CONV,<TRM>));

EXPR ETACONV();
     BEGIN NEW TRM,START;START←PC;
     IF CHECK('ETACONV) ∧ (TRM←TERM()) ∧ SC()
      THEN RETURN ETACONVSEM(TRM); PC←START;
     END;

EXPR ETACONVSEM(TRM);
     IF ISLAMBDATERM(TRM) ∧ ISAPPLTERM(MATRIXOF(TRM)) ∧
             EQ(BVAROF(TRM),ARGOF(MATRIXOF(TRM))) ∧
             NOTFRVT(BVAROF(TRM),FNOF(MATRIXOF(TRM)))
      THEN MKPROOFSTEP(MKWFF('?≡,TRM,FNOF(MATRIXOF(TRM))),
                             'NODEP,REASON('ETACONV,<TRM>))
      ELSE PRINTMES("NASTY ETACONV");

EXPR ALPHACONV();
     BEGIN NEW L1,TRM,V1,V2,START;START←PC;
     IF CHECK('ALPHACONV) ∧ (L1←NUMBER()) ∧ (V1←IDENT()) ∧ (V2←IDENT()) ∧ SC()
        THEN RETURN(ACONVSEM1(L1,V1,V2)); PC←START;
     IF CHECK('ALPHACONV) ∧ (TRM←TERM()) ∧ (V1←IDENT()) ∧ (V2←IDENT()) ∧ SC()
       THEN RETURN(ACONVSEM2(TRM,V1,V2)); PC←START;
     END;

EXPR ACONVSEM1(L1,V1,V2 :AWF,FS);
     IF ISPROOFSTEP(L1)
      THEN MKPROOFSTEP(MKWFF(RELOF(AWF←AWFFOF(L1)),FS←ACONV(FSTERMOF(AWF),V1,V2),
                IF EQUAL(FS,FSTERMOF(AWF)) THEN ACONV(SNTERMOF(AWF),V1,V2)
                       ELSE SNTERMOF(AWF)),
                   DEPOF(L1), REASON('ALPHACONV, <L1,V1,V2> ))
      ELSE PRINTMES("NASTY ALPHACONV");

EXPR ACONVSEM2(TRM,V1,V2);
     MKPROOFSTEP(MKWFF('?≡,TRM,ACONV(TRM,V1,V2)),'NODEP,REASON('ALPHACONV,<TRM,V1,V2>));
```

```
EXPR EQUIV();
    BEGIN NEW L1,L2,START; START←PC;
    IF CHECK('EQUIV) ∧ (L1←NUMBER()) ∧ (L2←NUMBER()) ∧ SC()
     THEN RETURN EQUIVSEM(L1,L2); PC←START;
    END;

EXPR EQUIVSEM(L1,L2:AWF1,AWF2);
    IF      ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2)
        ∧ ISLTAWFF(AWF1←AWFFOF(L1)) ∧ISLTAWFF(AWF2←AWFFOF(L2))
        ∧ EQUAL(FSTERMOF(AWF1), SNTERMOF(AWF2))
        ∧ EQUAL(FSTERMOF(AWF2), SNTERMOF(AWF1))
    THEN  MKPROOFSTEP(MKWFF('?≡,FSTERMOF(AWF1),SNTERMOF(AWF1)),
                    UNIONOF(DEPOF(L1),DEPOF(L2)),REASON('EQUIV,<L1,L2>))
    ELSE  PRINTMES("NASTY EQUIV");

EXPR REFL1();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('REFL1) ∧ (TRM←TERM()) ∧ SC()
     THEN RETURN REFL1SEM(TRM); PC←START;
    END;

EXPR REFL1SEM(TRM);
    MKPROOFSTEP(MKWFF('?≡,TRM,TRM), 'NODEP , REASON('REFL1,<TRM>));

EXPR REFL2();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('REFL2) ∧ (TRM←TERM()) ∧ SC()
     THEN RETURN REFL2SEM(TRM); PC←START;
    END;

EXPR REFL2SEM(TRM);
    MKPROOFSTEP(MKWFF('?⊂,TRM,TRM), 'NODEP , REASON('REFL2,<TRM>));

EXPR MIN1();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('MIN1) ∧ (TRM←TERM()) ∧ SC()
     THEN RETURN MIN1SEM(TRM); PC←START;
    END;

EXPR MIN1SEM(TRM);
    MKPROOFSTEP(MKWFF('?⊂,'UU,TRM),'NODEP , REASON('MIN1,<TRM>));

EXPR MIN2();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('MIN2) ∧ (TRM←TERM()) ∧ SC()
     THEN RETURN MIN2SEM(TRM); PC←START;
    END;

EXPR MIN2SEM(TRM);
    MKPROOFSTEP(MKWFF('?≡,MKAPPLTERM('UU,TRM),'UU),'NODEP , REASON('MIN2,<TRM>));

EXPR CONDT();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('CONDT) ∧ (TRM←CONDTERM()) ∧ SC()
```

```
     THEN RETURN CONDTSEM(TRM); PC←START;
    END;

EXPR CONDTSEM(TRM);
    IF ISTTCOND(TRM)
     THEN MKPROOFSTEP(MKWFF('?≡,TRM,TRUCASOF(TRM)),'NODEP , REASON('CONDT,<TRM>))
     ELSE PRINTMES("NASTY CONDT");

EXPR CONDF();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('CONDF) ∧ (TRM←CONDTERM()) ∧ SC()
     THEN RETURN CONDFSEM(TRM);PC←START;
    END;

EXPR CONDFSEM(TRM);
    IF ISFFCOND(TRM)
    THEN MKPROOFSTEP(MKWFF('?≡,TRM,FALCASOF(TRM)),'NODEP , REASON('CONDF,<TRM>))
    ELSE PRINTMES("NASTY CONDF");

EXPR CONDU();
    BEGIN NEW TRM,START; START←PC;
    IF CHECK('CONDU) ∧ (TRM←CONDTERM()) ∧ SC()
     THEN RETURN CONDUSEM(TRM); PC←START;
    END;

EXPR CONDUSEM(TRM);
    IF ISUUCOND(TRM)
     THEN MKPROOFSTEP(MKWFF('?≡,TRM,'UU),'NODEP , REASON('CONDU,<TRM>))
     ELSE PRINTMES("NASTY CONDU");

EXPR FIXP();
    BEGIN NEW L1,START;START←PC;
    IF CHECK('FIXP) ∧ (L1←NUMBER()) ∧ SC()
     THEN RETURN FIXPSEM(L1); PC←START;
    END;

EXPR FIXPSEM(L1 :AWF,MT,FIX,BV,MA);
    IF ISPROOFSTEP(L1) ∧ ISMUTERM(MT←(SNTERMOF(AWF←AWFFOF(L1)))) ∧
      ISFREEFORT(FIX←FSTERMOF(AWF),BV←BVAROF(MT),MA←MATRIXOF(MT))
       THEN RETURN(MKPROOFSTEP(MKWFF('?≡,FIX,SUBSTG(MA,FIX,BV)),
                   DEPOF(L1),REASON('FIXP,<L1>)))
       ELSE RETURN(PRINTMES("NASTY FIXP"));

EXPR SUBST();
    BEGIN NEW L1,N,L2,TRM,START;START←PC;
    IF CHECK('SUBST) ∧ (L1←NUMBER()) ∧ CHECK('OCC) ∧ (N←NUMBER())
               ∧ CHECK('IN) ∧ (L2←NUMBER()) ∧ SC()
      THEN RETURN SUBSTSEM1(L1,N,L2); PC←START;
    IF CHECK('SUBST) ∧ (L1←NUMBER()) ∧ CHECK('OCC) ∧ (N←NUMBER())
               ∧ CHECK('IN) ∧ (TRM←TERM()) ∧ SC()
      THEN RETURN SUBSTSEM2(L1,N,TRM); PC←START;
    END;

EXPR SUBSTSEM1(L1,N,L2);
```

```
     BEGIN NEW AWF1,AWF2,DEP;
     IF ISPROOFSTEP(L1) ∧ ISPROOFSTEP(L2) ∧ ISEQUIVAWFF(AWF1←AWFFOF(L1))
      THEN AWF2← AWFFOF(L2) ALSO
          DEP←UNIONOF(DEPOF(L1),DEPOF(L2)) ALSO
          RETURN MKPROOFSTEP(SUBW(AWF2,AWF1,N),DEP,
                         REASON('SUBST,<L1,'OCC,N,'IN,L2>))
      ELSE RETURN PRINTMES("NASTY SUBST");
     END;

EXPR SUBSTSEM2(L1,N,TRM);
     BEGIN  NEW AWF,REL,SNT;
     IF ISPROOFSTEP(L1)
      THEN AWF←AWFFOF(L1) ALSO REL←RELOF(AWF) ALSO
        SNT←SUBSTTT(TRM,SNTERMOF(AWF),FSTERMOF(AWF),N) ALSO
      RETURN MKPROOFSTEP(MKWFF(REL,TRM,SNT),DEPOF(L1),
                      REASON('SUBST,<L1,'OCC,N,'IN,TRM>))
      ELSE RETURN(PRINTMES("NASTY SUBST"));
     END;
```

APPENDIX 5

AUXILIARY COMMANDS

```
EXPR SHOW();
    BEGIN NEW N1,N2,START;
    START←PC;
    IF CHECK('SHOW) ∧ CHECK('LINE) ∧ (N1←NUMBER()) ∧
                  OPT(COLON() ∧ (N2←NUMBER())) ∧ SC()
      THEN RETURN SHOWSEM(N1,N2);
    PC←START;
    END;

EXPR SHOWSEM(N1,N2);
    BEGIN
    IF NULL(N2) THEN N2←N1;
    TERPRI(PRINC(TERPRI("  ")));
A;  IF(N1≤N2) THEN
      (IF ISPROOFSTEP(N1)
       THEN TERPRI(PRINTLINE(SEARCH(N1,PROOF))) ALSO N1←N1+1 ALSO GO A
       ELSE RETURN PRINTMES("NONEXISTING STEP"))
    ELSE RETURN PRINC("  ");
    END;

EXPR FETCH();
    BEGIN NEW ID, START;
    START←PC;
    IF CHECK('FETCH) ∧ (ID←IDENT()) ∧ SC() THEN RETURN FETCHSEM(ID);
    PC←START;
    END;

EXPR FETCHSEM(ID);
    INC(EVAL(<'INPUT,'FOO,'DSK?:>@<ID>),NIL);

EXPR CANCEL();
    BEGIN NEW N,START;  START←PC;
    IF CHECK('CANCEL) ∧ OPT(N←NUMBER()) ∧ SC()
      THEN RETURN CANCELSEM(N);
    PC←START;  END;

EXPR CANCELSEM(N);
    BEGIN
    IF NULL(N) THEN N←PFLENGTH;
    IF (N≤1)
       THEN (PFLENGTH←0)
       ALSO (PROOF←NIL)
       ALSO RETURN (PRINTMES("YOU HAVE DEMOLISHED YOUR PROOF"));
A;  IF (PFLENGTH LESSP N) THEN RETURN(PRINTM(PFLENGTH));
    PFLENGTH ←(PFLENGTH-1);
    PROOF←CDR PROOF;
    GO A;
    END;
```

APPENDIX 6

AUXILIARY FUNCTIONS

6.1 Predicates on Free and Bound Occurrences of Varibles on Terms, Awffs, etc.

```
EXPR NOTBNDVT(V,TRM);
    BEGIN
    IF ISIDENT(TRM) THEN RETURN T;
    IF ISAPPLTERM(TRM) THEN RETURN (NOTBNDVT(V,FNOF(TRM))∧
                              NOTBNDVT(V,ARGOF(TRM)));
    IF ISCONDTERM(TRM) THEN RETURN (NOTBNDVT(V,PREDOF(TRM))∧
                              NOTBNDVT(V,TRUCASOF(TRM))∧
                              NOTBNDVT(V,FALCASOF(TRM)));
    IF(ISLAMBDATERM(TRM) v ISMUTERM(TRM))
       THEN (IF EQ(BVAROF(TRM),V) THEN RETURN NIL
            ELSE RETURN NOTBNDVT(V,MATRIXOF(TRM)));
    END;

EXPR BOUNDV(V,TRM); ¬NOTBNDVT(V,TRM);

EXPR NOTFRVT(V,TRM);
    BEGIN
        IF ISAPPLTERM(TRM) THEN RETURN (NOTFRVT(V,FNOF(TRM))∧NOTFRVT(V,ARGOF(TRM)));
        IF ISCONDTERM(TRM) THEN RETURN (NOTFRVT(V,PREDOF(TRM)) ∧
                                  NOTFRVT(V,TRUCASOF(TRM)) ∧
                                  NOTFRVT(V,FALCASOF(TRM)));
         IF ISLAMBDATERM(TRM) v ISMUTERM(TRM)
            THEN RETURN (EQ(V,BVAROF(TRM)) v NOTFRVT(V,MATRIXOF(TRM)));
        RETURN( ¬EQ(V,TRM));
    END;

EXPR FREEV(V,TRM); (¬NOTFRVT(V,TRM));

EXPR NOTFRVW(V,WF);
    IF EMPTY(WF) THEN  T
    ELSE NOTFRVT(V,FSTERMOF(FSTOF(WF))) ∧
         NOTFRVT(V,SNTERMOF(FSTOF(WF))) ∧
         NOTFRVW(V,RMDR(WF));

EXPR NOTFREE(V,LN);
    IF EMPTY(LN) THEN T ELSE
    (IF NOTFRVW(V,WFFOF(FSTOF(LN))) THEN NOTFREE(V,RMDR(LN)));

EXPR ISFREEFORT(X,V,TRM);
       BEGIN
       IF ISIDENT(TRM) THEN RETURN T;
       IF ISAPPLTERM(TRM) THEN RETURN ISFREEFORT(X,V,FNOF(TRM))∧
                                ISFREEFORT(X,V,ARGOF(TRM));
       IF ISCONDTERM(TRM) THEN RETURN ISFREEFORT(X,V,PREDOF(TRM))∧
                                ISFREEFORT(X,V,TRUCASOF(TRM)) ∧
                                ISFREEFORT(X,V,FALCASOF(TRM)) ;
```

```
      IF ISLAMBDATERM(TRM) v ISMUTERM(TRM) THEN
         IF EQ(V,BVAROF(TRM)) v FREEV(BVAROF(TRM),X) THEN RETURN NIL
              ELSE RETURN ISFREEFORT(X,V,MATRIXOF(TRM));
      END;

EXPR ISFREEFORW(X,V,WF);
      IF EMPTY(WF) THEN T
       ELSE ISFREEFORT(X,V,FSTERMOF(FSTOF(WF))) ∧
           ISFREEFORT(X,V,SNTERMOF(FSTOF(WF))) ∧
           ISFREEFORW(X,V,RMDR(WF));
```

6.2 Miscellaneous Functions Used in INCL. CUT. CASES. SHOW

```
EXPR PICKUP(WF,N);
    IF EQ(N,1) THEN <FSTOF(WF)> ELSE PICKUP(RMDR(WF),N-1);

EXPR INCLTEST(LN,WF);
    BEGIN
    IF EMPTY(LN) THEN RETURN(T);
    IF TESTM(WFFOF(FSTOF(LN)),WF) THEN RETURN(INCLTEST(RMDR(LN),WF));
    END;

EXPR TESTM(WF1,WF2);
    IF EMPTY(WF1) THEN T
    ELSE MEMBER(FSTOF(WF1),WF2) ∧ TESTM(RMDR(WF1),WF2);

EXPR TESTCASES(LN1,LN2,LN3,TRM);
    TESTC(MKWFF('?≡,TRM,'TT),LN1) ∧
    TESTC(MKAWF('?≡,TRM,'UU),LN2) ∧
    TESTC(MKAWF('?≡,TRM,'FF),LN3);

EXPR TESTC(WF,LN);
    IF EMPTY(LN) THEN NIL ELSE
       IF EQUAL(WF,WFFOF(FSTOF(LN))) THEN T
          ELSE TESTC(WF,RMDR(LN));

EXPR FIND(LN,TRM1,TRM2);
    IF EMPTY(LN) THEN NIL ELSE
    IF EQUAL(MKWFF('?≡,TRM1,TRM2),WFFOF(FSTOF(LN)))
    THEN FSTOF(LN) ELSE FIND(RMDR(LN),TRM1,TRM2);

EXPR REMOVE(LN,N);
    IF EQ(LN,NIL) THEN NIL ELSE
    (IF EQ(N,FSTOF(LN)) THEN RMDR(LN)
    ELSE (FSTOF(LN) CONS REMOVE(RMDR(LN),N)));

EXPR OPT(X);
    IF X THEN X ELSE T;
```

6.3 Conversion and Substitution Routines

```
EXPR CONVT(TRM);
    BEGIN NEW BV,MAS,MA,FNEW;
    IF ISIDENT(TRM) THEN RETURN TRM;
    IF ISCONDTERM(TRM) THEN RETURN MKCONDTERM(CONVT(PREDOF(TRM)),
                    CONVT(TRUCASOF(TRM)),CONVT(FALCASOF(TRM)));
    IF ISLAMBDATERM(TRM) THEN RETURN MKLAMBDATERM(BVAROF(TRM),CONVT(MATRIXOF(TRM)));
    IF ISMUTERM(TRM) THEN RETURN MKMUTERM(BVAROF(TRM),CONVT(MATRIXOF(TRM)));
    IF ISAPPLTERM(TRM) THEN
          (IF ISLAMBDATERM(FNOF(TRM))
           THEN BV←BVAROF(FNOF(TRM))
           ALSO MA←MATRIXOF(FNOF(TRM))
           ALSO MAS←SUBSTG(MA,CONVT(ARGOF(TRM)),BV)
           ALSO RETURN IF EQUAL(MA,MAS) THEN TRM ELSE
                 CONVT(MAS)
           ELSE RETURN IF ISLAMBDATERM(FNEW←CONVT(FNOF(TRM))) THEN
             CONVT(MKAPPLTERM(FNEW,CONVT(ARGOF(TRM))))
              ELSE MKAPPLTERM(FNEW,CONVT(ARGOF(TRM))));
    END;

EXPR SUBSTG(TRM,X,V1);
    BEGIN
    IF ISIDENT(TRM) ∧ EQ(TRM,V1) THEN RETURN X;
    IF ISIDENT(TRM)  THEN RETURN TRM;
    IF ISAPPLTERM(TRM) THEN RETURN MKAPPLTERM(SUBSTG(FNOF(TRM),X,V1),
                                    SUBSTG(ARGOF(TRM),X,V1));
    IF ISCONDTERM(TRM) THEN RETURN MKCONDTERM(SUBSTG(PREDOF(TRM),X,V1),
                                    SUBSTG(TRUCASOF(TRM),X,V1),
                                    SUBSTG(FALCASOF(TRM),X,V1));
    IF ISLAMBDATERM(TRM)
       THEN RETURN (IF EQ(V1,BVAROF(TRM)) ∨ FREEV(BVAROF(TRM),X)
                 THEN TRM
                 ELSE MKLAMBDATERM(BVAROF(TRM),SUBSTG(MATRIXOF(TRM),X,V1)));
    IF ISMUTERM(TRM)
       THEN RETURN (IF EQ(V1,BVAROF(TRM)) ∨ FREEV(BVAROF(TRM),X)
                 THEN TRM
                 ELSE MKMUTERM(BVAROF(TRM),SUBSTG(MATRIXOF(TRM),X,V1)));
    END;

EXPR ACONV(TRM,V1,V2:X);
    BEGIN
    IF NOTBNDVT(V2,TRM) THEN RETURN TRM;
    IF ISCONDTERM(TRM) THEN BEGIN
     IF BOUNDV(V2,PREDOF(TRM)) THEN RETURN MKCONDTERM(ACONV(PREDOF(TRM),V1,V2),
                                     TRUCASOF(TRM),FALCASOF(TRM));
     IF BOUNDV(V2,TRUCASOF(TRM)) THEN RETURN MKCONDTERM(PREDOF(TRM),
                         ACONV(TRUCASOF(TRM),V1,V2),FALCASOF(TRM));
     IF BOUNDV(V2,FALCASOF(TRM)) THEN RETURN MKCONDTERM(PREDOF(TRM),
                         TRUCASOF(TRM),ACONV(FALCASOF(TRM),V1,V2));END;
    IF ISAPPLTERM(TRM) ∧ BOUNDV(V2,FNOF(TRM))
     THEN RETURN MKAPPLTERM(ACONV(FNOF(TRM),V1,V2),ARGOF(TRM));
    IF ISAPPLTERM(TRM)
     THEN RETURN MKAPPLTERM(FNOF(TRM),ACONV(ARGOF(TRM),V1,V2));
    IF ISLAMBDATERM(TRM) ∧ EQ(V2,BVAROF(TRM))
     THEN RETURN (IF FREEV(V1,MATRIXOF(TRM)) ∨
                 EQUAL(X←SUBSTG(MATRIXOF(TRM),V1,V2),MATRIXOF(TRM))
                 THEN TRM
```

```
                    ELSE MKLAMBDATERM(V1,X));
       IF ISLAMBDATERM(TRM)
        THEN RETURN MKLAMBDATERM(BVAROF(TRM),ACONV(MATRIXOF(TRM),V1,V2));
       IF ISMUTERM(TRM) ∧ EQ(V2,BVAROF(TRM))
        THEN RETURN (IF FREEV(V1,MATRIXOF(TRM)) ∨
                    EQUAL(X←SUBSTG(MATRIXOF(TRM),V1,V2),MATRIXOF(TRM))
                    THEN TRM
                    ELSE MKMUTERM(V1,X));
       IF ISMUTERM(TRM)
        THEN RETURN MKMUTERM(BVAROF(TRM),ACONV(MATRIXOF(TRM),V1,V2));
       END;

   EXPR SUBW(AWF1,AWF2,N);
       BEGIN NEW TRM1,TRM2;
       SUBCOUNT←N;
       TRM1←DOSUBST(FSTERMOF(AWF1),SNTERMOF(AWF2),FSTERMOF(AWF2));
       TRM2←(IF EQ(SUBCOUNT,C) THEN SNTERMOF(AWF1)
              ELSE DOSUBST(SNTERMOF(AWF1),SNTERMOF(AWF2),FSTERMOF(AWF2)));
       RETURN MKWFF(RELOF(AWF1),TRM1,TRM2);
       END;

   EXPR SUBSTTT(TRM1,TRM2,TRM3,N);
       BEGIN
       SUBCOUNT←N;
       RETURN DOSUBST(TRM1,TRM2,TRM3);
       END;

   EXPR DOSUBST(TRM1,TRM2,TRM3);
       BEGIN NEW AUX1,AUX2,AUX3;
       IF EQUAL(TRM1,TRM3) THEN (SUBCOUNT←SUBCOUNT-1) ALSO
         (IF EQ(SUBCOUNT,0) THEN RETURN TRM2 ELSE RETURN TRM1);
       IF ISIDENT(TRM1) THEN RETURN TRM1;
       IF ISCONDTERM(TRM1) THEN
          AUX1←DOSUBST(PREDOF(TRM1),TRM2,TRM3) ALSO
          AUX2←(IF EQ(SUBCOUNT,0) THEN TRUCASOF(TRM1)
                 ELSE DOSUBST(TRUCASOF(TRM1),TRM2,TRM3)) ALSO
          AUX3←(IF EQ(SUBCOUNT,0) THEN FALCASOF(TRM1)
                 ELSE DOSUBST(FALCASOF(TRM1),TRM2,TRM3)) ALSO
          RETURN MKCONDTERM(AUX1,AUX2,AUX3);
       IF ISAPPLTERM(TRM1) THEN
          AUX1←DOSUBST(FNOF(TRM1),TRM2,TRM3) ALSO
          AUX2←(IF EQ(SUBCOUNT,0) THEN ARGOF(TRM1)
                 ELSE DOSUBST(ARGOF(TRM1),TRM2,TRM3)) ALSO
          RETURN MKAPPLTERM(AUX1,AUX2);
       IF ISLAMBDATERM(TRM1) ∨ ISMUTERM(TRM1) THEN
          IF FREEV(BVAROF(TRM1),TRM2) ∨ FREEV(BVAROF(TRM1),TRM3) THEN
             RETURN TRM1 ELSE RETURN
             (IF ISLAMBDATERM(TRM1)
                 THEN MKLAMBDATERM(BVAROF(TRM1),DOSUBST(MATRIXOF(TRM1),TRM2,TRM3))
                 ELSE MKMUTERM(BVAROF(TRM1),DOSUBST(MATRIXOF(TRM1),TRM2,TRM3)));
       END;
```

APPENDIX 7

MANIPULATION OF THE DATA STRUCTURE

7.1 Constructors

```
EXPR MKCONDTERM(PR,TC,FC); ('?!COND CONS PR CONS TC CONS FC);

EXPR MKAPPLTERM(FN,ARG); ('?!APPLY CONS FN CONS ARG);

EXPR MKLAMBDATERM(V,TRM); ('?!LAMBDA CONS V CONS TRM);

EXPR MKMUTERM(V,TRM); ('?!MU CONS V CONS TRM);

EXPR MKAWF(X,Y,Z); (X CONS Y CONS Z);

EXPR MKWFF(X,Y,Z); <(X CONS Y CONS Z)>;

EXPR MKPROOFSTEP(X,Y,Z);IF EQ(Y,'NODEP) THEN <X,NIL,Z> ELSE <X,Y,Z>;

EXPR REASON(X,Y);(X CONS Y);
```

7.2 Selectors

```
EXPR PREDOF(TRM); CADR TRM ;

EXPR TRUCASOF(TRM); CADDR TRM ;

EXPR FALCASOF(TRM); CDDDR TRM ;

EXPR DEPOF(X:P);  BEGIN P←SEARCH(X,PROOF);RETURN(P[3]);END;

EXPR RELOF(X); CAR X;

EXPR FSTERMOF(X); CADR X;

EXPR SNTERMOF(X); CDDR X;

EXPR AWFFOF(X); (CAR  WFFOF(X));

EXPR WFFOF(X:P);  BEGIN P←SEARCH(X,PROOF); RETURN( P[2]);END;

EXPR FSTOF(X);  CAR X ;

EXPR RMDR(X);  CDR X ;

EXPR FNOF(X);CADR X;
EXPR ARGOF(X); CDDR X;

EXPR BVAROF(X); CADR X;

EXPR MATRIXOF(X); CDDR X;
```

7.3 Predicates

```
EXPR ISEQUIVAWFF(AWF);    EQ(RELOF(AWF),'?≡);

EXPR ISLTAWFF(AWF);       EQ(RELOF(AWF),'?⊂);

EXPR ISINCL(N,WF);        (LNT(WF)≥N):

EXPR ISTTCOND(TRM);       EQ(PREDOF(TRM),'TT);

EXPR ISFFCOND(TRM);       EQ(PREDOF(TRM),'FF);

EXPR ISUUCOND(TRM);       EQ(PREDOF(TRM),'UU);

EXPR ISPROOFSTEP(L);      (PFLENGTH≥L);

EXPR EMPTY(X);            EQ(X,NIL);

EXPR ISLINE(X);           ¬(ATOM(X));

EXPR ISIDENT(X);          ATOM(X);

EXPR ISAPPLTERM(TRM); EQ((CAR TRM),'?!APPLY);

EXPR ISCONDTERM(TRM); EQ((CAR TRM), '?!COND );

EXPR ISLAMBDATERM(TRM); EQ((CAR TRM) ,'?!LAMBDA);

EXPR ISMUTERM(TRM); EQ((CAR TRM), '?!MU);
```

7.4 Miscellaneous Functions

```
EXPR UNIONOF(LN1,LN2);
    BEGIN
    IF EQ(LN1,'NODEP) ∨ EQ(LN1,NIL) THEN RETURN LN2;
    IF EQ(LN2,'NODEP) ∨ EQ(LN2,NIL) THEN RETURN LN1;
    IF MEMQ((CAR LN1),LN2) THEN RETURN(UNIONOF((CDR LN1),LN2))
                    ELSE RETURN((CAR LN1) CONS (UNIONOF((CDR LN1),LN2)));
    END;

EXPR UNIONW(WF1,WF2);
    IF EQUAL(WF1,NIL) THEN WF2 ELSE
    (IF MEMBER((CAR WF1),WF2) THEN UNIONW((CDR WF1),WF2)
        ELSE ((CAR WF1) CONS UNIONW((CDR WF1),WF2)));

EXPR ADDLINE(X);
    BEGIN    PFLENGTH ← PFLENGTH + 1;
             PROOF←((PFLENGTH CONS X) CONS PROOF);    END;

EXPR SEARCH(X, P);
    IF EQ(P[1,1],X) THEN P[1] ELSE SEARCH(X,(CDR P));

EXPR LNT(X);
    IF EQ((CDR X),NIL) THEN 1 ELSE (LNT(CDR X) + 1);

EXPR SUBWV(WF,X,V:FS);
      IF EQ(WF,NIL) THEN NIL ELSE
        (MKAWF(RELOF(FS←FSTERMOF(WF)),SUBSTG(FSTERMOF(FS),X,V),
         SUBSTG(SNTERMOF(FS),X,V)) CONS SUBWV(RMDR(WF),X,V));
```

INDEX

In this index all the functions appearing in the program are listed in alphabetic order. Each name is followed by the number of the appendix where the function is defined.